Chapter 12

Give Me Arrays

In This Chapter

- Storing multiple variables in an array
- Creating an array
- Understanding character arrays
- Sorting values in an array
- Working with multidimensional arrays
- Sending an array to a function

When I first learned to program, I avoided the topic of arrays. They didn't make sense to me. Arrays sport their own methods and madness, which is different from working with single variables in C. Rather than shun this topic and skip ahead to the next chapter (which isn't any easier), consider embracing the array as a lovely, weird, and useful tool.

Behold the Array

In the real world, information comes individually or in groups. You may find a penny on the road and then a nickel and maybe a quarter! To handle such fortunes in the C language, you need a way to gather variables of a similar type into groups. A row of variables would be nice, as would a queue. The word used in C is *array*.

Avoiding arrays

At some point in your programming career, an array becomes inevitable. As an example, consider Listing 12-1. The code asks for and displays your three top scores, presumably from a game.

Listing 12-1: High Scores, the Awful Version

```
#include <stdio.h>

int main()
{
    int highscore1,highscore2,highscore3;

    printf("Your highest score: ");
    scanf("%d",&highscore1);
    printf("Your second highest score: ");
    scanf("%d",&highscore2);
    printf("Your third highest score: ");
    scanf("%d",&highscore3);

    puts("Here are your high scores");
    printf("#1 %d\n",highscore1);
    printf("#2 %d\n",highscore2);
    printf("#3 %d\n",highscore3);

    return(0);
}
```

The code in Listing 12-1 asks for three integer values. Input is stored in the three `int` variables declared in Line 5. Lines 15 through 17 display the values. Simple.

Exercise 12-1: Type the source code from Listing 12-1 into the editor. Build and run.

Typing that code can be a lot of work, right? You could just copy and paste things, which makes it easier. But typing the code isn't the problem.

Exercise 12-2: Modify the source code from Listing 12-1 so that the fourth-highest score is added. Build and run.

Now imagine the coding you would have to do if the user requested to see their third-highest score. Yep, it gets messy fast. Things work better, of course, when you use arrays.

Understanding arrays

An *array* is series of variables of the same type: a dozen `int` variables, two or three `double` variables, or a string of `char` variables. The array doesn't contain all the same values. No, it's more like a series of cubbyholes into which you stick different values.

An array is declared like any other variable. It's given a type and a name and then also a set of square brackets. The following statement declares the `highscore` array:

```
int highscore[];
```

This declaration is incomplete; the compiler doesn't yet know how many items, or *elements*, are in the array. So if the `highscore` array were to hold three elements, it would be declared like this:

```
int highscore[3];
```

This array contains three elements, each of them its own `int` value. The elements are accessed like this:

```
highscore[0] = 750;
highscore[1] = 699;
highscore[2] = 675;
```

An array element is referenced by its index number in square brackets. The first item is index 0, which is something you have to remember. In C, you start counting at 0, which has its advantages, so don't think it's stupid.

In the preceding example, the first array element, `highscore[0]`, is assigned the value `750`; the second element, `699`; and the third, `675`.

After initialization, an array variable is used like any other variable in your code:

```
var = highscore[0];
```

This statement stores the value of array element `highscore[0]` to variable `var`. If `highscore[0]` is equal to 750, `var` is equal to 750 after the statement executes.

Exercise 12-3: Rewrite the source code from your solution to Exercise 12-2 using an array as described in this section — but keep in mind that your array holds four values, not three.

Many solutions exist for Exercise 12-3. The brute-force solution has you stuffing each array variable individually, line after line, similar to the source code in Listing 12-1. A better, more insightful solution is offered in Listing 12-2.

Listing 12-2: High Scores, a Better Version

```
#include <stdio.h>

int main()
{
    int highscore[4];
    int x;

    for(x=0;x<4;x++)
    {
        printf("Your #%d score: ",x+1);
        scanf("%d",&highscore[x]);
    }

    puts("Here are your high scores");
    for(x=0;x<4;x++)
        printf("#%d %d\n",x+1,highscore[x]);

    return(0);
}
```

Most of the code from Listing 12-2 should be familiar to you, albeit the new array notation. The `x+1` arguments in the `printf()` statements (Lines 10 and 16) allow you to use the *x* variable in the loop but display its value starting with 1 instead of 0. Although C likes to start numbering at 0, humans still prefer starting at 1.

Exercise 12-4: Type the source code from Listing 12-2 into your editor and build a new project. Run it.

Though the program's output is pretty much the same as the output in Exercises 12-2 and 12-3, the method is far more efficient, as proven by working Exercise 12-5:

Exercise 12-5: Modify the source code from Listing 12-2 so that the top ten scores are input and displayed.

Imagine how you'd have to code the answer to Exercise 12-5 if you chose not to use arrays!

- The first element of an array is 0.
- When declaring an array, use the full number of elements, such as `10` for ten elements. Even though the elements are indexed from `0` through `9`, you still must specify `10` when declaring the array's size.

Initializing an array

As with any variable in C, you can initialize an array when it's declared. The initialization requires a special format, similar to this statement:

```
int highscore[] = { 750, 699, 675 };
```

The number in the square brackets isn't necessary when you initialize an array, as shown in the preceding example. That's because the compiler is smart enough to count the elements and configure the array automatically.

Exercise 12-6: Write a program that displays the stock market closing numbers for the past five days. Use an initialized array, `marketclose[]`, to hold the values. The output should look something like this:

```
Stock Market Close
Day 1: 14450.06
Day 2: 14458.62
Day 3: 14539.14
Day 4: 14514.11
Day 5: 14452.06
```

Exercise 12-7: Write a program that uses two arrays. The first array is initialized to the values `10`, `12`, `14`, `15`, `16`, `18`, and `20`. The second array is the same size but not initialized. In the code, fill the second array with the square root of each of the values from the first array. Display the results.

Playing with character arrays (strings)

You can create an array using any of the C language's standard variable types. A `char` array, however, is a little different: It's a string.

As with any array, you can declare a `char` array initialized or not. The format for an initialized `char` array can look like this:

```
char text[] = "A lovely array";
```

As I mention elsewhere, the array size is calculated by the compiler, so you don't need to set a value in the square brackets. Also — and most importantly — the compiler adds the final character in the string, a null character: `\0`.

You can also declare the array as you would declare an array of values, though it's kind of an insane format:

```
char text[] = { 'A', ' ', 'l', 'o', 'v', 'e', 'l', 'y',
                ' ', 'a', 'r', 'r', 'a', 'y', '\0' };
```

Each array element in the preceding line is defined as its own `char` value, including the `\0` character that terminates the string. No, I believe that you'll find the double quote method far more effective at declaring strings.

The code in Listing 12-3 plods through the `char` array one character at a time. The *`index`* variable is used as, well, the index. The `while` loop spins until the `\0` character at the end of the string is encountered. A final `putchar()` function (in Line 14) kicks in a newline.

Listing 12-3: Displaying a char Array

```
#include <stdio.h>

int main()
{
    char sentence[] = "Random text";
    int index;

    index = 0;
    while(sentence[index] != '\0')
    {
        putchar(sentence[index]);
        index++;
    }
    putchar('\n');
    return(0);
}
```

Exercise 12-8: Type the source code from Listing 12-3 into your editor. Build and run the program.

The `while` loop in Listing 12-3 is quite similar to most string display routines found in the C library. These functions probably use pointers instead of arrays, which is a topic unleashed in Chapter 18. Beyond that bit o' trivia, you could replace Lines 8 through 14 in the code with the line

```
puts(sentence);
```

or even with this one:

```
printf("%s\n",sentence);
```

When the `char` array is used in a function, as shown in the preceding line, the square brackets aren't necessary. If you include them, the compiler believes that you screwed up.

Working with empty char arrays

Just as you can declare an empty, or uninitialized, `float` or `int` array, you can create an empty `char` array. You must be precise, however: The array's size must be 1 greater than the maximum length of the string to account for that NULL character. Also, you have to ensure that whatever input fills the array doesn't exceed the array's size.

In Listing 12-4, the `char` array `firstname` at Line 5 can hold 15 characters, plus 1 for the `\0` at the end of the string. That 15-character limitation is an assumption made by the programmer; most first names are fewer than 15 characters long.

Listing 12-4: Filling a char Array

```
#include <stdio.h>

int main()
{
    char firstname[16];

    printf("What is your name? ");
    fgets(firstname,16,stdin);
    printf("Pleased to meet you, %s\n",firstname);
    return(0);
}
```

An `fgets()` function in Line 8 reads in data for the `firstname` string. The maximum input size is set to 16 characters, which already accounts for the null character because `fgets()` is smart that way. The text is read from `stdin`, or standard input.

Exercise 12-9: Create a new project using the source code from Listing 12-4. Build and run, using your first name as input.

Try running the program again, but fill up the buffer: Type more than 15 characters. You'll see that only the first 15 characters are stored in the array. Even the Enter key press isn't stored, which it would be otherwise when input is fewer than 15 characters.

Exercise 12-10: Modify your source code from Exercise 12-9 so that the program also asks for your last name, storing that data in another array. The program should then greet you by using both your first and last names.

Yes, the Enter key press is stored as part of your name, which is how input is read by the `fgets()` function. If your first name is Dan, the array looks like this:

```
firstname[0] == 'D'
firstname[1] == 'a'
firstname[2] == 'n'
firstname[3] == '\n'
firstname[4] == '\0'
```

That's because input in C is stream oriented, and Enter is part of the input stream as far as the `fgets()` function is concerned. You can fix this issue by obeying Exercise 12-11.

Exercise 12-11: Rewrite your source code from Exercise 12-10 so that the `scanf()` function is used to read in the first and last name strings.

Of course, the problem with the `scanf()` function is that it doesn't check to ensure that input is limited to 15 characters — that is, unless you direct it to do so:

Exercise 12-12: Modify the `scanf()` functions in your source code from Exercise 12-11 so that the conversion character used is written as `%15s`. Build and run.

The `%15s` conversion character tells the first `scanf()` function to read only the first 15 characters of input and place it into the `char` array (string). Any extra text is then read by the second `scanf()` function, and any extra text after that is discarded.

It's critical that you understand streaming input when it comes to reading text in C. Chapter 13 offers additional information on this important topic.

Sorting arrays

Computers are designed to quickly and merrily accomplish boring tasks, such as sorting an array. In fact, they love doing it so much that "the sort" is a basic computer concept upon which many theories and algorithms have been written. It's a real snoozer topic if you're not a Mentat or a native of the planet Vulcan.

The simplest sort is the *bubble sort*, which not only is easy to explain and understand but also has a fun name. It also best shows the basic array-sorting philosophy, which is to swap values between two elements.

Suppose that you're sorting an array so that the smallest values are listed first. If `array[2]` contains the value `20`, and `array[3]` contains the value `5`, these two elements would need to swap values. To make it happen, you use a temporary variable in a series of statements that looks like this:

```
temp=array[2];        /* Save 20 in temp */
array[2]=array[3];    /* Store 5 in array[2] */
array[3]=temp;        /* Put 20 in array[3] */
```

In a bubble sort, each array element is compared with every other array element in an organized sequence. When one value is larger (or smaller) than another, the values are swapped. Otherwise, the comparison continues, plodding through every possible permutation of comparisons in the array. Listing 12-5 demonstrates.

Listing 12-5: A Bubble Sort

```
#include <stdio.h>

#define SIZE 6

int main()
{
    int bubble[] = { 95, 60, 6, 87, 50, 24 };
    int inner,outer,temp,x;

/* Display original array */
    puts("Original Array:");
    for(x=0;x<SIZE;x++)
        printf("%d\t",bubble[x]);
    putchar('\n');

/* Bubble sort */
    for(outer=0;outer<SIZE-1;outer++)
    {
        for(inner=outer+1;inner<SIZE;inner++)
        {
            if(bubble[outer] > bubble[inner])
            {
                temp=bubble[outer];
                bubble[outer] = bubble[inner];
                bubble[inner] = temp;
            }
        }
    }

/* Display sorted array */
    puts("Sorted Array:");
    for(x=0;x<SIZE;x++)
        printf("%d\t",bubble[x]);
    putchar('\n');

    return(0);
}
```

Listing 12-5 is long, but it's easily split into three parts, each headed by a comment:

- Lines 10 through 14 display the original array.
- Lines 16 through 28 sort the array.
- Lines 30 through 34 display the sorted array (duplicating Lines 10 through 14).

The constant `SIZE` is defined in Line 3. This directive allows you to easily change the array size in case you reuse this code again later (and you will).

The sort itself involves nested `for` loops: an outer loop and an inner loop. The outer loop marches through the entire array, one step at a time. The inner loop takes its position one element higher in the array and swoops through each value individually.

Exercise 12-13: Copy the source code from Listing 12-5 into your editor and create a new project, ex1213. Build and run.

Exercise 12-14: Using the source code from Listing 12-5 as a starting point, create a program that generates 40 random numbers in the range from 1 through 100 and stores those values in an array. Display that array. Sort that array. Display the results.

Exercise 12-15: Modify the source code from Exercise 12-14 so that the numbers are sorted in reverse order, from largest to smallest.

Exercise 12-16: Write a program that sorts the text in the 21-character string "C Programming is fun!"

Change an array's size, will you?

When an array is declared in C, its size is set. After the program runs, you can neither add nor remove more elements. So if you code an array with 10 elements, as in

```
int topten[10];
```

you cannot add an 11th element to the array. Doing so leads to all sorts of woe and misery.

To use nerdy lingo, an array in C is not *dynamic*: It cannot change size after the size has been established. Other programming languages let you resize, or *redimension,* arrays. C doesn't.

Multidimensional Arrays

The arrays described in the first part of this chapter are known as *single-dimension* arrays: They're basically a series of values, one after the other. That's fine for describing items that march single file. When you need to describe items in the second or third dimension, you conjure forth a multidimensional type of array.

Making a two-dimensional array

It helps to think of a two-dimensional array as a grid of rows and columns. An example of this type of array is a chess board — a grid of 8 rows and 8 columns. Though you can declare a single 64-element array to handle the job of representing a chess board, a two-dimensional array works better. Such a thing would be declared this way:

```
int chess[8][8];
```

The two square brackets define two different dimensions of the `chess` array: 8 rows and 8 columns. The square located at the first row and column would be referenced as `chess[0][0]`. The last square on that row would be `chess[0][7]`, and the last square on the board would be `chess[7][7]`.

In Listing 12-6, a simple tic-tac-toe board is created using a two-dimensional matrix: 3-by-3. Lines 9 through 11 fill in the matrix. Line 12 adds an X character in the center square.

Listing 12-6: Tic-Tac-Toe

```
#include <stdio.h>

int main()
{
    char tictactoe[3][3];
    int x,y;

/* initialize matrix */
    for(x=0;x<3;x++)
        for(y=0;y<3;y++)
            tictactoe[x][y]='.';
    tictactoe[1][1] = 'X';

/* display game board */
    puts("Ready to play Tic-Tac-Toe?");
```

(continued)

Listing 12-6 *(continued)*

```
    for(x=0;x<3;x++)
    {
        for(y=0;y<3;y++)
            printf("%c\t",tictactoe[x][y]);
        putchar('\n');
    }
    return(0);
}
```

Lines 14 through 21 display the matrix. As with its creation, the matrix is displayed by using a nested `for` loop.

Exercise 12-17: Create a new project using the source code shown in Listing 12-6. Build and run.

A type of two-dimensional array that's pretty easy to understand is an array of strings, as shown in Listing 12-7.

Listing 12-7: An Array of Strings

```
#include <stdio.h>

#define SIZE 3

int main()
{
    char president[SIZE][8] = {
        "Clinton",
        "Bush",
        "Obama"
    };
    int x,index;

    for(x=0;x<SIZE;x++)
    {
        index = 0;
        while(president[x][index] != '\0')
        {
            putchar(president[x][index]);
            index++;
        }
        putchar('\n');
    }
    return(0);
}
```

Line 7 in Listing 12-7 declares a two-dimensional `char` array: `president`. The first value in square brackets is the number of items (strings) in the array. The second value in square brackets is the maximum size required to hold the largest string. The largest string is `Clinton` with seven letters, so eight characters are required, which includes the terminating `\0` or null character.

Because all items in the array's second dimension must have the same number of elements, all strings are stored using eight characters. Yep, that's wasteful, but it's the way the system works. Figure 12-1 illustrates this concept.

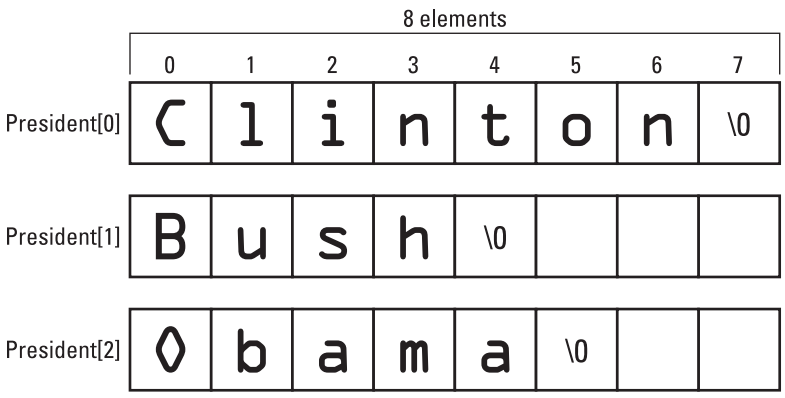

Figure 12-1: Storing strings in a two-dimensional array.

Exercise 12-18: Type the source code from Listing 12-7 into your editor; build and run the program.

Lines 16 through 22 in Listing 12-7 are inspired by Exercise 12-8, earlier in this chapter. The statements basically plod through the `president` array's second dimension, spitting out one character at a time.

Exercise 12-19: Replace Lines 15 through 23 in Listing 12-7 with a single `puts()` function to display the string. Here's how that statement looks:

```
puts(president[x]);
```

When working with string elements in an array, the string is referenced by the first dimension only.

Exercise 12-20: Modify your source code from Exercise 12-19 so that three more presidents are added to the array: Washington, Adams, and Jefferson.

Going crazy with three-dimensional arrays

Two-dimensional arrays are pretty common in the programming realm. Multidimensional is insane!

Well, maybe not. Three- and four-dimensional arrays have their place. The big deal is that your human brain has trouble keeping up with the various possible dimensions.

Listing 12-8 illustrates code that works with a three-dimensional array. The declaration is found at Line 5. The third dimension is simply the third set of square brackets, which effectively creates a 3D tic-tac-toe game board.

Listing 12-8: Going 3D

```
#include <stdio.h>

int main()
{
    char tictactoe[3][3][3];
    int x,y,z;

/* initialize matrix */
    for(x=0;x<3;x++)
        for(y=0;y<3;y++)
            for(z=0;z<3;z++)
                tictactoe[x][y][z]='.';
    tictactoe[1][1][1] = 'X';

/* display game board */
    puts("Ready to play 3D Tic-Tac-Toe?");
    for(z=0;z<3;z++)
    {
        printf("Level %d\n",z+1);
        for(x=0;x<3;x++)
        {
            for(y=0;y<3;y++)
                printf("%c\t",tictactoe[x][y][z]);
            putchar('\n');
        }
    }
    return(0);
}
```

Lines 8 through 12 fill the array with data, using variables *x*, *y*, and *z* as the three-dimensional coordinates. Line 13 places an `X` character in the center cube, which gives you an idea of how individual elements are referenced.

The rest of the code from Lines 15 through 26 displays the matrix.

Exercise 12-21: Create a three-dimensional array program using the source code from Listing 12-8. Build and run.

Lamentably, the output is two-dimensional. If you'd like to code a third dimension, I'll leave that up to you.

Declaring an initialized multidimensional array

The dark secret of multidimensional arrays is that they don't really exist. Internally, the compiler still sees things as single dimensions — just a long array full of elements. The double (or triple) bracket notation is used to calculate the proper offset in the array at compile time. That's okay because the compiler does the work.

You can see how multidimensional arrays translate into regular old boring arrays when you declare them already initialized. For example:

```
int grid[3][4] = {
    5, 4, 4, 5,
    4, 4, 5, 4,
    4, 5, 4, 5
    };
```

The grid array consists of three rows of four items each. As just shown, it's declared as a grid and it looks like a grid. Such a declaration works, as long as the last element doesn't have a comma after it. In fact, you can write the whole thing like this:

```
int grid[3][4] = { 5, 4, 4, 5, 4, 4, 5, 4, 4, 5, 4, 5 };
```

This statement still defines a multidimensional array, but you can see how it's really just a single-dimension array with dual indexes. In fact, the compiler is smart enough to figure out the dimensions even when you give only one of them, as in this example:

```
int grid[][4] = { 5, 4, 4, 5, 4, 4, 5, 4, 4, 5, 4, 5 };
```

In the preceding line, the compiler sees the 12 elements in an array grid, so it automatically knows that it's a 3-by-4 matrix based on the `4` in the brackets. Or you can do this:

```
int grid[][6] = { 5, 4, 4, 5, 4, 4, 5, 4, 4, 5, 4, 5 };
```

In this example, the compiler would figure that you have two rows of six elements. But the following example is just wrong:

```
int grid[][] = { 5, 4, 4, 5, 4, 4, 5, 4, 4, 5, 4, 5 };
```

The compiler isn't going to get cute. In the preceding line, it sees an improperly declared single-dimension array. The extra square brackets aren't needed.

Exercise 12-22: Rewrite the code from Exercise 12-17 so that the tic-tac-toe game board is initialized when the array is declared — including putting the X in the proper spot.

Arrays and Functions

Creating an array for use inside a function works just like creating an array for use inside the `main()` function: The array is declared, it's initialized, and its elements are used. You can also pass arrays to and from functions, where the array's elements can be accessed or manipulated.

Passing an array to a function

Sending an array off to a function is pretty straightforward. The function must be prototyped with the array specified as one of the arguments. It looks like this:

```
void whatever(int nums[]);
```

This statement prototypes the `whatever()` function. That function accepts the integer array `nums` as its argument. The entire array — every element — is passed to the function, where it's available for fun and frolic.

When you call a function with an array as an argument, you must omit the square brackets:

```
whatever(values);
```

In the preceding line, the `whatever()` function is called using the array `values` as an argument. If you keep the square brackets, the compiler

assumes that you meant only to pass a single element and that you forgot to specify which one. So this is good:

```
whatever(values[6]);
```

But this is not good:

```
whatever(values[]);
```

The code shown in Listing 12-9 features the `showarray()` function that eats an array as an argument. It's a `void` function, so it doesn't return any values, but it can manipulate the array.

Listing 12-9: Mr. Function, Meet Mr. Array

```
include <stdio.h>

#define SIZE 5

void showarray(int array[]);

int main()
{
    int n[] = { 1, 2, 3, 5, 7 };

    puts("Here's your array:");
    showarray(n);
    return(0);
}

void showarray(int array[])
{
    int x;

    for(x=0;x<SIZE;x++)
        printf("%d\t",array[x]);
    putchar('\n');
}
```

The `showarray()` function is called at Line 12. See how the `n` array is passed without its angle brackets? Remember that format!

At Line 16, the `showarray()` function is declared with the array specified using square brackets, just like the prototype at Line 5. Within the function, the array is accessed just like it would be in the `main()` function, which you can see at Line 21.

Exercise 12-23: Type the source code from Listing 12-9 into your editor. Build and run the program to ensure that it works.

Exercise 12-24: Add a second function, `arrayinc()`, to your source code from Exercise 12-23. Make it a `void` function. The function takes an array as its argument. The function adds 1 to each value in the array. Have the `main()` function call `arrayinc()` with array `n` as its argument. Then call the `showarray()` function a second time to display the modified values in the array.

Returning an array from a function

In addition to being passed an array, a function in C can return an array. The problem is that arrays can be returned only as pointers. (This topic is covered in Chapter 19.) But that's not the worst part:

In Chapter 19, you discover the scandalous truth that C has no arrays — that they are merely cleverly disguised pointers. (Sorry to save that revelation for the end of this chapter.) Array notation does have its place, but pointers are where the action is.